Attachment

Thank you for your valuable and constructive comments! Due to space constraints, we provide more supplementary information in this attachment. Specifically, Attachment-A provides more details about the bug-finding cases mentioned in Shared-Q2 (shared by Reviewer#B-Q2 and Reviewer#C-Q1). Attachment-B provides detailed evaluation results for each target contract, as mentioned in Reviewer#A-A1. Attachment-C provides more details about the differences in the benchmark contract set, as mentioned in Reviewer#B-A1.

Attachment-A. Cases for Bug-Finding

The Case of BNB

```
// The source code of the BNB smart contract
(0xB8c77482e45F1F44dE1745F52C74426C631bDD52)
/*
    Allow another contract to spend some tokens on your behalf
*/
function approve(address _spender, uint256 _value)
    returns (bool success) {
    if (_value <= 0) throw;
    allowance[msg.sender][_spender] = _value; // Does not emit an Approval event
    return true;
}
```

As shown in the above figure, the BNB contract in our benchmark contract set does not emit the `Approval` event in the implementation of the `approve` function. However, the `Approval` event is a "MUST" requirement in the ERC-20 Specification (https://eips.ethereum.org/EIPS/eip-20). This non-compliance with the ERC-20 specification by the BNB implementation may lead to unexpected behavior when interacting with the BNB contract using off-chain programs or browser wallet plugins. It is considered a vulnerability according to the common ERC20 vulnerability classification list (https://github.com/sec-bit/awesome-buggy-erc20-tokens/blob/master/ERC20_token_issue_list.md#b7-no-approval).

```
// Example implementation of ERC-20 compatible smart contracts

    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256
value);

   function approve(address spender, uint256 amount) public virtual override
returns (bool) {
        _approve(_msgSender(), spender, amount);
        return true;
    }
    /**
```

```
     * @dev Sets `amount` as the allowance of `spender` over the `owner` s
tokens.
     *
     * This internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    function _approve(address owner, address spender, uint256 amount) internal
virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount); // Emit an Approval Event
    }
```

During SolMigrator's test case migration process, the test cases migrated from existing ERC-20 compatible source contracts will check if the Approval event is emitted during execution. Therefore, these test cases will fail on the BNB contract. Developers can further investigate the cause of these failures and determine whether the BNB contract needs to emit the Approval Event in the `approve` function.

The Case of PepeToken

```
// The source code of the PepeToken smart contract
(0x6982508145454Ce325dDbE47a25d4ec3d2311933)
/**
 * @dev See {IERC20-transfer}.
 *
 * Requirements:
 *
 * - `recipient` cannot be the zero address.
 * - the caller must have a balance of at least `amount`.
 */
function transfer(address recipient, uint256 amount) public virtual override
returns (bool) {
    _transfer(_msgSender(), recipient, amount);
    return true;
}

function setRule(bool _limited, address _uniswapV2Pair, uint256
_maxHoldingAmount, uint256 _minHoldingAmount) external onlyOwner {
    limited = _limited;
    uniswapV2Pair = _uniswapV2Pair;
    maxHoldingAmount = _maxHoldingAmount;
    minHoldingAmount = _minHoldingAmount;
}
```

```
/**
 * @dev Moves `amount` of tokens from `sender` to `recipient`.
 *
 * This internal function is equivalent to {transfer}, and can be used to
 * e.g. implement automatic token fees, slashing mechanisms, etc.
 *
 * Emits a {Transfer} event.
 *
 * Requirements:
 *
 * - `sender` cannot be the zero address.
 * - `recipient` cannot be the zero address.
 * - `sender` must have a balance of at least `amount`.
 */
function _transfer(
    address sender,
    address recipient,
    uint256 amount
) internal virtual {
    require(sender != address(0), "ERC20: transfer from the zero address");
    require(recipient != address(0), "ERC20: transfer to the zero address");

    _beforeTokenTransfer(sender, recipient, amount);

    uint256 senderBalance = _balances[sender];
    require(senderBalance >= amount, "ERC20: transfer amount exceeds balance");
    unchecked {
        _balances[sender] = senderBalance - amount;
    }
    _balances[recipient] += amount;

    emit Transfer(sender, recipient, amount);

    _afterTokenTransfer(sender, recipient, amount);
}

function _beforeTokenTransfer(
    address from,
    address to,
    uint256 amount
) override internal virtual {
    require(!blacklists[to] && !blacklists[from], "Blacklisted");

    if (uniswapV2Pair == address(0)) {
        require(from == owner() || to == owner(), "trading is not started");
        return;
    }

    if (limited && from == uniswapV2Pair) {
        require(super.balanceOf(to) + amount <= maxHoldingAmount &&
super.balanceOf(to) + amount >= minHoldingAmount, "Forbid");
    }
}
```

As shown in the above figure, the PepeToken smart contract in our benchmark set disable token transfers by default, and the contract owner can enable or disable them at any time. After the smart contract is deployed, the contract owner needs to call the `setRule` function to enable transfers. When a user attempts to transfer their tokens by calling the `transfer` function, the `transfer` function will call the `_beforeTokenTransfer` function to check if transfers have been enabled by the owner: if not, the transfer operation will be reverted. Additionally, during the operation of the contract, the contract owner can enable/disable transfers at any time by calling the `setRules` function.

However, a recent study (*CRPWarner: Warning the Risk of Contract-related Rug Pull in DeFi Smart Contracts*, TSE'24) indicates that the implementation that allows the owner to enable and disable token transfers between users at any time, may lead to the risk of contract-related rug pulls. They refer to this defect as a *Limiting Sell Order*. Because if transfers between users are unintentionally or maliciously disabled, users will be unable to sell their tokens, leading to financial losses directly.

In other ERC20 contracts, there is no additional requirement to enable transfers, so they generate test cases that attempt to transfer tokens directly without any additional operations. During SolMigrator's migration process, direct transfer operations will be reverted by the PepeToken contract, causing these test cases to fail on the PepeToken contract. Developers can further analyze these failures to determine whether the implementation that allows the owner to arbitrarily disable or enable transfers is a bug or an intentional design choice.

Attachment-B. Function Coverage of the Migrated Test Cases (vs. Real Historical Transactions)

To address Reviewer#A's Question 1, we further analyzed the function coverage of the migrated test cases and provided more empirical results in this attachment. Specifically, we analyzed the migrated test cases for each target-source migration pair and recorded the functions they covered. Then, we compared the functions covered by the migrated test cases with those covered by real-world transactions of the target contract. In line with Section VI.D, for each target contract, we chose the migration source that achieves the highest function coverage, along with the test cases migrated from it, for comparison with real-world transactions. Since developers typically perform test migrations between very similar contracts, selecting the most suitable source contract better simulates real-world scenarios.

The following two tables show the results for ERC20 and ERC721 categories, respectively. The first column is the target contract that the test cases are migrated to, and the second column is the migration source. The third column shows the proportion of functions covered by the migrated test cases relative to those invoked by real-world transactions. The fourth column shows the proportion of real-world transactions that called these covered transactions.

Target	Source	Function Coverage (vs. Historical Txns) (%)	Real Historical Transactions Calling the Covered Functions(%)
A0	A9	55.6%	99.2%
A1	A6	80%	99.9%
A2	A3	100.0%	100.0%
A3	A4	50.0%	53.7%

Target	Source	Function Coverage (vs. Historical Txns) (%)	Real Historical Transactions Calling the Covered Functions(%)
A4	A3	66.7%	96.7%
A5	A4	54.5%	97.5%
A6	A4	57.1%	99.7%
A7	A1	62.5%	99.6%
A8	A5	57.1	2.8%
A9	A0	66.7%	99.9%

Target	Source	Function Coverage (vs. Historical Txns) (%)	Real Historical Transactions Calling the Covered Functions(%)
B0	B1	80.0%	89.1%
B1	B5	70.0%	94.1%
B2	B6	36.4%	17.0%
B3	B7	85.7%	11.4%
B4	B0	66.7%	1.7%
B5	B1	83.3%	99.7%
B6	B2	75%	99.7%
B7	B2	71.4%	28.8%
B8	B7	58.3%	20.0%
B9	B3	80.0%	99.9%

Our results indicate that, **on average, the migrated test cases can cover 67.9% of the functions actually used in real-world transactions of the target contract**. **For 95% (19 out of 20) of the target smart contracts, the migrated test cases can cover more than 50% of the functions used in real-world transactions**. Furthermore, the differences between columns 3 and 4 indicate that the functions covered by SolMigrator are more commonly used in historical transactions than those not covered.

It is also important to note that, **similar to existing migration tools [7,40,47], SolMigrator is not designed to maximize the code coverage of the target smart contracts**. This is because, in the context of text migration, our primary goal is to migrate *common* usage patterns from existing smart contracts, and these common usages often concentrate on a limited set of key functionalities.

Attachment-C. Differences Among Benchmark Smart Contracts

To answer Reviewer#B-Q1, we carefully reviewed the benchmark contract set and found no true duplicates, i.e., duplicates at source-code level with only parameter differences. We found that although the benchmark smart contracts belong to the same categories, i.e., ERC20/ERC721, they actually have diverse implementations.

```
// The source code of the GraphToken smart contract
(0xc944e90c64b2c07662a292be6244bdf05cda44a7)
/**
 * @dev Mint new tokens.
 * @param _to Address to send the newly minted tokens
 * @param _amount Amount of tokens to mint
 */
function mint(address _to, uint256 _amount) external onlyMinter {
    _mint(_to, _amount);
}

function _mint(address account, uint256 amount) internal virtual {
    require(account != address(0), "ERC20: mint to the zero address");

    _beforeTokenTransfer(address(0), account, amount);

    _totalSupply = _totalSupply.add(amount);
    _balances[account] = _balances[account].add(amount);
    emit Transfer(address(0), account, amount);
}

/**
 * @dev Return if the `_account` is a minter or not.
 * @param _account Address to check
 * @return True if the `_account` is minter
 */
function isMinter(address _account) public view returns (bool) {
    return _minters[_account];
}
```

For example, regarding mint-related functions, ERC20 tokens such as BNB do not allow token minting. In contrast, tokens like GraphToken do allow minting, i.e., users with specific roles (such as the Minter role in GraphToken) can mint tokens for other addresses by calling the `mint` function.

```
// The source code of the TetherToken smart contract
(0xdac17f958d2ee523a2206206994597c13d831ec7)
// Issue a new amount of tokens
// these tokens are deposited into the owner address
//
// @param _amount Number of tokens to be issued
function issue(uint amount) public onlyOwner {
    require(_totalSupply + amount > _totalSupply);
    require(balances[owner] + amount > balances[owner]);

    balances[owner] += amount;
    _totalSupply += amount;
    Issue(amount);
}
```

As shown in the figure above, the mint logic of TetherToken is different yet again from that of GraphToken: it only allows the owner to mint tokens for themselves and does not support minting tokens for other addresses.

In addition to our manual inspection, **the analysis of true negatives in Section VI.C also indicates the functional differences in the benchmark contracts**. For 68.1% of true negatives, there are no existing counterparts in the target contract, directly preventing SolMigrator from matching test cases from the source contract to the target contract. This indicates that the source contract contains functionality that the target contract does not have. In the remaining 31.9%, the test cases from the source contract can be matched to the target contract, but they fail to execute in the target contract due to functional differences between the source and target contracts. This indicates that the functionality provided by both the target and source contracts actually has different implementations. Both of these cases suggest functional differences in the benchmark contracts.